AFRL-RI-RS-TR-2009-235
Final Technical Report
October 2009

ACCOUNTABILITY FOR INFORMATION FLOW VIA EXPLICIT FORMAL PROOF

Carnegie Mellon University

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2009-235 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/
JAMES SIDORAN
Work Unit Manager

/s/
WARREN H. DEBANY, Jr.
Technical Advisor, Information Grid Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
OCTOBER 2009	Final	May 2007 – May 2009

4. TITLE AND SUBTITLE

ACCOUNTABILITY FOR INFORMATION FLOW VIA EXPLICIT FORMAL PROOF

5a. CONTRACT NUMBER
N/A

5b. GRANT NUMBER
FA8750-07-2-0028

5c. PROGRAM ELEMENT NUMBER
N/A

6. AUTHOR(S)

Frank Pfenning, Lujo Bauer, Peter Lee, Michael K. Reiter, and Brian Witten

5d. PROJECT NUMBER
NICE

5e. TASK NUMBER
00

5f. WORK UNIT NUMBER
07

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Carnegie Mellon University
Department of Computer Science
5000 Forbes Avenue
Pittsburgh, PA 15213-3815

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/RIGA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
N/A

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2009-235

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Logical techniques have been developed that capture both authorization and information flow requirements in security applications. These logical techniques achieve a significantly higher degree of end-to-end accountability in distributed systems than is currently possible. Furthermore, a case study has shown that these techniques are applicable to security policies that are relevant to the needs of the intelligence community while providing much greater flexibility in security policy specification. A prototype implementation has demonstrated the practicality of the proposed file system architecture. Symantec, the industrial partner in the project, is presently pursuing a significant related business opportunity.

15. SUBJECT TERMS
Information Flow Requirements, Security Policy Specification, Formal Logic, Temporal Logic, Proof Carrying Code

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			James L. Sidoran
U	U	U	UU	24	19b. TELEPHONE NUMBER (*Include area code*) N/A

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

TABLE OF CONTENTS

1 SUMMARY

Logical techniques that capture both authorization and information flow requirements in security applications have been developed. These techniques achieve a significantly higher degree of end-to-end accountability in distributed systems than is currently possible. Furthermore, it has been demonstrated that these techniques are applicable to security policies that are relevant to the needs of the intelligence community while providing much greater flexibility in security policy specification. Key is the development of a logic that allows policy statements by principals and explicit time to coexist harmoniously.

The cornerstone of the approach is the notion of a *formal proof,* which can be audited by a human and checked automatically by machine, augmented with cryptographic primitives, which provide a different form of evidence for basic logical judgments. Applications lie in security for general distributed information infrastructures. To test the practicality of this approach, a prototype file system using formal proofs has been implemented called PCFS (for *proof-carrying file system*). Evaluation of the file system has shown that it is both efficient and modular through a new organization that separates a front end with proof generation and proof checking from a back end that uses only cryptographic capabilities. A specific case study shows how realistic security policies derived from intelligence-community needs can be specified logically, and then implemented automatically to provide flexible, secure, and accountable access to sensitive files.

The project also laid the foundation for a technology transition to Symantec Corporation, which is currently in progress.

2 INTRODUCTION

For the purpose of security, the access that principals (users, programs, etc.) have to resources (databases, files, etc.) is usually restricted. This practice, generically called *access control*, is pervasive; its use ranges from low-level memory subsystems to file systems and web servers. Despite differences in both the resources protected and principles from whom they are protected, the high level architecture of most access control mechanisms is similar: all calls that access a protected resource pass through a subsystem called the *reference monitor*, which, based on the identity of the principal making the call, the resource being accessed, and the nature of the call (read, write, create, etc.) either allows the call to proceed or blocks it.

A significant question in the design of an access control subsystem is how the reference monitor decides which requests to authorize and which to deny. The traditional solution to this problem is to represent the *access control policy* as a take that for every principal *k*, resource *r*, and operation *o* tells whether principal *k* may perform operation *o* on resource *r*. This representation is called an *access control matrix*, usually implemented through *access control lists* (ACLs) associated directly with resources. Although access control lists and matrices are both simple to implement and widely used in practice, they are extremely low level and suffer from the drawback that they not carry information about *why* a certain access is allowed or denied. This limits their use in scenarios where accountability of access is a concern (e.g., we would like to know why an individual was able to read a file not merely that she was able to). This is important in military and intelligence servers with classified information, for businesses that have proprietary data to protect, and in matters of customer privacy. The second problem with access control matrices and lists is that it is difficult to keep them up to date with changing access requirements, and this often results in policy errors and inadvertent access. We illustrate the limitations of access control matrices and lists in the following example.

Example. Consider a hypothetical scenario where Alice is an employee of the company AuthCo, and within the company world for the team GovTeam, which handles contracts from the government. As a member of the team, Alice has access to a government dataset *d*. Owing to the sensitive nature of the dataset, this access is contingent upon her maintaining a government security clearance. Suppose that the access control policy for the dataset *d* is represented using ACLs. While Alice has access, her name would be on the ACL of *d*. Observe, however, that the access control list does not provide any evidence as to why Alice has this access (it does not record her affiliation with GovTeam nor her security clearance). As a result, if an internal auditor were to try to determine whether it is legitimate to have Alice's name on the ACL or not, he would have to consult many other sources. Further, if Alice were to lose her government security clearance, some administrator would have to manually observe this change and remove her name from the ACL. If, for any reason, the administrator failed to take notice or to act promptly, Alice would continue to have access when she should not, resulting in a security breach.

The problems with ACLs, as illustrated by the example, can be eliminated using a *rule-based representation* of policies. The policy is represented as a set of cause and effect rules, and access is allowed only if it is entailed by the rules. As an illustration, the policy in the example may be expressed by the following rules.

1. For any principal *k*, if *k* works for GovTeam and *k* has a government security clearance, then *k* can read dataset *d*.
2. Alice works for GovTeam.
3. Alice has a government security clearance.

The main advantage of representing the policy as rules is that the reason for granting access becomes explicit. Here, for instance, was an audit to be performed, it would be clear that Alice has access because she works for GovTeam and also has a government security clearance. Moreover, if access is granted only if explicit evidence for it is provided or inferred, this evidence may be logged, increasing accountability and improving assurance in the access control subsystem. The second advantage of using rules is that such a representation ensures that the consequences of the policy change automatically change with conditions. For example, if Alice were to lose her government security clearance, there would no longer be any inference to authorize Alice's read request, and hence she would no longer be able to read *d*.

The next relevant issue is determining a formal language that may be used to represent policy rules and determine their consequences. This work rests on the idea that *formal logic* may be used to represent policy rules and enforce them. As an example, the policy rules (1)-(3) may be represented by the three formulas below, assuming that the predicate worksFor(*k*, GovTeam) means that *k* works for GovTeam, hasClearance(*k*) means that *k* has government security clearance, and may(*k*, *d*, read) means that *k* is allowed to read dataset *d*.

1. $\forall k$. ((worksFor(*k*, GovTeam) hasClearance(*k*)) may(*k*, *d*, read)
2. worksFor(Alice, GovTeam)
3. hasClearance(Alice)

Now it is easy to check that (1)-(3) logically entail the formula may(Alice, *d*, read). The use of logic for policy representation is not a mere convenience, but very natural and pragmatic.

- As illustrated above, the cause and effect nature of policy rules make it natural to *express* them in logic.
- Once represented in logic, the consequence of the policy rules and unambiguous since they are defined by the logic's semantics. Hence logic provides a rigorous foundation for *defining* the meanings of policies.
- A logical proof that shows why policy rules authorize access may be logged, improving the *accountability* of the access control subsystem.
- Logical inference and automatic proof search based on it can be used to *implement* the policy rules directly.

Having established the central role of logic for reducing errors in policy administration and improving accountability for access control decisions, three further questions arise which form the main threads of this project.

1. Which access control policies and high-level policy motifs are prevalent in the intelligence community?
2. Which forms of logical expression should be directly available in the logic so that policies from the intelligence community can be represented?
3. Can we design and implement a practical file system that exploits logic-based authorization?

The remainder of the reports is organized along our answers to these questions. The findings strongly support the contention that logic-based authorization is practical, and could be of great value to the intelligence community in significantly reducing policy administration errors and accountability for information flow.

3 METHODS, ASSUMPTIONS, AND PROCEDURES

3.1 Development of Relevant Policy Problems

We coupled decades of Symantec internal experience working with the intelligence community, including decades of experience by individuals who were previously government service members of the intelligence community, with dozens of more recent meetings with intelligence community stakeholders in the process of formulating policy on handling of classified and sensitive but unclassified information. These meetings included meetings with individuals from NSA, CIA, and DIA, along with other intelligence community leaders such as members of the Office of the Director of National Intelligence (O-DNI). Additionally, researchers from Symantec met with intelligence divisions of military commands such as Joint Forces Command, as well as Defense Contractors such as Lockheed Martin, actively involved in several intelligence agency programs along with other executive branch government departments such as the Department of Homeland Security.

The results of these meetings are summarized in five internal policy reports delivered by Symantec (Serenyi & Witten 2008) acting as a subcontractor, to Carnegie Mellon University. Another source of information is Executive Orders of the White House (1995, 2003) and the Director of Central Intelligence (1995, 1998). We then synthesized these reports and additional information into a formal, logical theory in the authorization logic described in a technical report (Garg et al. 2009).

3.2 Foundations for Logics of Affirmation and Knowledge

Although many authorization policies may be represented in propositional or first-order logic, there are some commonly occurring policy idioms that are best represented with specialized logical connectives? We use the term *authorization logic* to designate any logic that has been designed with the explicit purpose of representing authorization policies. One particularly important connective, written "*k* says *s*" for a principal *k* and sentence *s*, expresses that principal *k* says, claims, or supports the formula *s*, without forcing *s* to be necessarily true. As the following continued example illustrates, it is extremely important for representing authority of principals on parts of policies, and for capturing the interaction between rules created by different principals.

Example (continued). Let us assume there are three principals involved in determining authorization: (a) Admin, who has ultimate authority on deciding who should have access to the dataset *d*, (b) AuthCoHr which determines team affiliations of employees of AuthCo, and (c) Gov, which certifies government security clearances. The principal who certifies (creates) each rule is indicated at the beginning of the rule in square brackets [-].

1. [Admin] for every principal *k*, if AuthCoHr certifies that *k* works for GovTeam and Gov certifies that *k* has a government security clearance, the *k* is allowed to read dataset *d*.
2. [AuthCoHr] Alice works for GovTeam.
3. [Gov] Alice has a government security clearance.

Using the connective *k* says *s*; these rules may be represented as follows:

1. Admin says *k*. (((AuthCoHr says worksFor(*k*, GovTeam)) (Gov says hasClearance(*k*))) may (*k*, *d*, read))
2. AuthCoHr says worksFor(Alice, GovTeam)
3. Gov says HasClearance(Alice)

In general, Principal Alice will be allowed to read dataset *d* only if there is a proof of the following formula: Admin says may (Alice, *d*, read). This is the case in the developed authorization logic.

Our method has been to analyze various proposed laws of logical reasoning with the "says" connective that are both semantically sound and in accord with its policy interpretation as in the examples above.

3.3 Proof-Carrying Authorization

Proof-carrying authorization (PCA) is a rigorous mechanism based in cryptography and formal proofs that is used for *distributed enforcement* of authorization policies represented in logic. It is based on two central ideas:

- Principal *k* should be allowed to perform operation *o* on resource *r* only if *k* can produce a *formal proof object* which shows that the policy rules in effect entail that access should be allowed. For instance, in our running example, *k* would have to prove the formula Admin says may(*k*, *r*, *o*) from the policy formulas (1)-(3).
- Policy rules can be established a priori using digital signatures: if principal *k* signs the formula *s* with its private key, then the resulting digital certificate is evidence that *k* says *s* holds.

Based on these ideas, PCA allows distributed enforcement of authorization policies represented in logic in the following manner. Administrators sign policy rules in digital certificates which are then published through any mechanism. A principal *k* wishing to access, selects certificates it believes relevant to its authorization, and, taking the policy rules instated by the certificates as hypotheses, construction a logical proof *M* which establishes that it has legitimate access. Along with its request to perform access, *k* also provides the proof *M* and the certificates used in it to the reference monitor (hence the adjective "proof-carrying"). The reference monitor verifies the digital certificates by checking the digital signatures in them, and also verifies the logical proof *M*. If both checks succeed, the access is allowed, otherwise it is blocked.

While this work follows this method to a large extent, PCA as sketched above it too inefficient for direct use in a file system, where access may be frequent and speed is essential. In a more distributed setting, however, the above has been shown to be successful (Bauer et al. 2007, Bauer et al. 2008).

4 RESULTS AND DISCUSSION

4.1 Access Control for Classified Information in Logical Form

Analysis of the access control policies in the intelligence community has led to the following findings regarding the design of authorization logic:

- The "says" connective, which is used to represent certificates digitally signed by principals, should be weak in deductive power in order to limit inadvertent consequences of policies.
- The logic should be *intuitionistic*, which is logic of explicit evidence. Using an intuitionistic logic means that the reasons for why access was granted are clear and direct, because intuitionistic logic prohibits the rule of indirect proof.
- The logic should be *first-order* rather than *higher-order*. First-order logics are easier for proof search and to establish meta-theoretic properties of the logic itself and of policies encoded in it. The added expressive power of higher-order logics does not seem to be required in practice and might indeed lead to unintended consequences.
- The logic should incorporate *interpreted predicates* whose truth or falsity is decided at time of access based on the current system state. In particular, interpreted predicates should have access to meta-data associated with a file. This is important to capture the life cycle of sensitive information, which includes stages as a working paper, a classified, or an unclassified document.
- The logic should integrate a concept of *absolute time*, because many policies we found are dynamic and require explicit time bounds. For example, a working paper must be classified, declassified, deleted, or reinstated as a working paper within 90 days from its creation. Another example is background checks necessary for security clearance, which expire after 5, 10, or 15 years, depending on the level of clearance and kind of check performed.

Once an authorization logic with these characteristics has been defined (as is detailed in the next section), then it is straightforward for a researcher to cast informal policy rules into the authorization logic. Detailed policy rules, together with an explanation of their informal meaning, can be found in an extensive technical report (Garg et al. 2009).

4.2 An Authorization Logic with Explicit Time

An authorization logic, called BL, that responds to the findings regarding access control policies in the previous section is presented in a technical report (Garg 2008), refining an earlier design (DeYoung et al. 2008). First, BL contains all the usual connectives and quantifiers from intuitionistic predicate logic. Second, BL contains the modality *k* says *s*, which means that principal *k* claims the truth of formula *s*, without necessarily forcing *s* to be true. In practice, the modality is used to distinguish policy rules and credentials created by different individuals or sources of authority. That is, a certificate containing sentence *s* digitally signed by *k* is represented on the logical side precisely as the formula *k* says *s*. BL also supports a fixed hierarchy of principals, primarily in order to represent local authorities and commonly shared assumption. We write $k' \geq k$ if *k'* is stronger than *k*, so *k* claims everything that *k'* does.

Logically, the modality can be characterized by the following laws:

- If |- *s* then |- *k* says *s*. This is the modal rule of necessitation, for each principal.
- |- (*k* says (*s* r)) (*k* says *s* *k says r*)
- |- (*k* says *s*) (*k'* says (*k* says *s*))
- |- (*k'* says s) (*k* says *s*) if $k' \geq$ k

It is also important to see the laws that are not validated. Most importantly

- Not |- (*k* says *s*) *s*.
- Not |- *s* (*k* says *s*).

The first expresses that just saying (digitally signing) something does not make it true. The second expresses that we are not forced to affirm something which is true but depends on assumptions (some of which we might not share), while the rule of necessitation forces us to accept what is logically valid, without possibly contentious assumptions. The finding is that this balance of laws provides an excellent foundation for security policies in the intelligence community in neither being too strong (which might admit too many accesses) nor too weak (which might not admit enough).

Support for explicit time in BL is manifest in the modality *s* @ [*u*, *w*], which means that *s* is true during the time interval [*u*, *w*], but possibly not outside of it. The variables *u* and *w* both denote time points, encoded as integers that count seconds from a fixed point of reference. This modality is very useful for representing policies that expire at stipulated points of time, as well as those that use time relatively (for example, allowing access for 90 days from an event such as a file creation).

Explicit time is useful for determining consequences of policies in practice only in conjunction with methods for reasoning about inequality between time points. For example, if Alice has a certificate that allows her a certain access from January 1, 2007 to December 31, 2009, it is only reasonable that she be able to derive from it a proof that allows her access on August 12, 2009. Constructing such a proof requires the ability to reason that August 12, 2009 lies between January 1, 2007 and December 31, 2010. To this end, BL includes special formulas called *constraints*, on particular form of which may be inequalities on time points, $u \leq w$. Constraints differ from ordinary predicates in that they are not established by hypothesis; instead their verification relies on an external constraint solver which is formally embedded in the logic via a satisfaction judgment |= *c*, where *c* denotes a constraint.

Besides explicit time, many real authorization policies use the state of the system as an input. The state may represent the progress of a workflow or a protocol. As a simple example, the authorization policy for a homework directory in a class administration system may allow read and write access for the teaching assistants while the homework is being prepared, read and write access for students while the homework may be submitted, and read access for teaching assistants after submissions are closed. A simple way to model the different stages – preparation, submission, and post-submission – may be as a state system; the stage may be written by the instructor as an attribute (meta-data) on the homework directory, and the access policy rules may be contingent upon the value of the attribute. To incorporate such elements of state in policy rules, BL allows interpreted predicates, whose truth is not justified by the logical hypotheses, but

by an external solver that refers to the state of the system. In the case of our example here, the solver would check the value of the attribute on the homework directory. From a proof-theoretic perspective, constraints and interpreted predicates are similar. We maintain a syntactic distinction between the two in BL because they are enforced differently in the proof-carrying file system (PCFS).

Here are some policy idioms that can be expressed in BL. The simplest use of time is to represent certificate expiration. For example, if Alice signs a certificate allowing Bob read-access to the file *secret.txt* from February 1, 2009 to February 28, 2009, this can be represented in BL as the formula (Alice says (may Bob *secret.txt* read)) @ [2009:02:01,2009:02:28]. The interaction of the @ connective with constraints in BL ensures that this time interval is respected during enforcement.

The second crucial policy idiom is expiration of access rights not necessarily tied to certificates. As an example, consider a so-called *working paper* in the intelligence community. Working papers are marked by an extended file attribute *status* which is set to working (*u*), where *u* is the time of file creation. The following, extracted from the case study of the intelligence community, expresses that *k* may read the file *f*, if the owner *k'* of *f* affirms that this is the case, and the file has been a working paper for at most 90 days.

- Admin says (has-xattr(*f*, status, working(*u*)) owner(*f*, *k'*) *k'* says may(*k*, *f*, read) may(*k*, *f*, read)) @ [*u*, *u*+90d]

BL satisfies the kind of metatheoretic properties, which show that it is well-defined as logic. In particular, its natural deduction formulation satisfies normalization, and its sequent calculus formulation satisfied cut elimination. Interested readers may refer to a technical report for additional details (Garg 2008). One property that it does *not* satisfy is decidability. Fortunately, in the context of our main intended application, a proof-carrying file system, decidability is not required. What is, however, required is that proofs expressed in the logic can be verified with relative efficiency. That is indeed the case – some measurements can be found in the next section.

4.3 A Proof-Carrying File System

The *proof-carrying file system* (PCFS) has been implemented as a testbed for logic-based authorization in practice. It is freely available (PCFS 2009) and described in detail in a technical report (Garg & Pfenning 2009). It builds on ideas from proof-carrying authorization (PCA). It is currently implemented as a local file system for Linux servers, but its architecture has been designed to support distribution. The name PCFS is an acronym for Proof-Carrying File System, even though access requests in PCFS *do not* carry proofs as they do in proof-carrying authorization. Instead proof verification is offlined to trusted verifiers that are invoked prior to file access.

Briefly, PCFS works as follows. The access policy is represented as logical formulas in BL and distributed to users in the form of digital certificates signed by policy administrators. A user constructs formal proofs, which show that the policy entails certain permissions for her. Each proof is checked by a trusted proof verifier, which gives the user a signed capability in return. This capability, called a *procap* (for *pro*ven *cap*ability), can be used repeatedly to authorize access to file system operations; the file system checks the procap each time it is required for authorization. Therefore, policy enforcement in PCFS follows the path:

Policy → Proof → Procap → File Access

Figure 1 shows the PCFS architecture. Numbers are used to label steps in order in which they occur in practice. Steps 1–6 deal with the logic, and include proof generation, proof verification, and creation of procaps. These steps are performed in advance of file access, and happen infrequently (usually when a user accesses a file for the first time). Once procaps are stored, they can be used repeatedly to perform file operations (steps 7–12). The solid black vertical line in the diagram separates parts that happen in user space, that is, before and after a file system call (left side of the line) from those that take place during a file system call (right side of the line). In the following we describe the steps of Figure 1 in some detail.

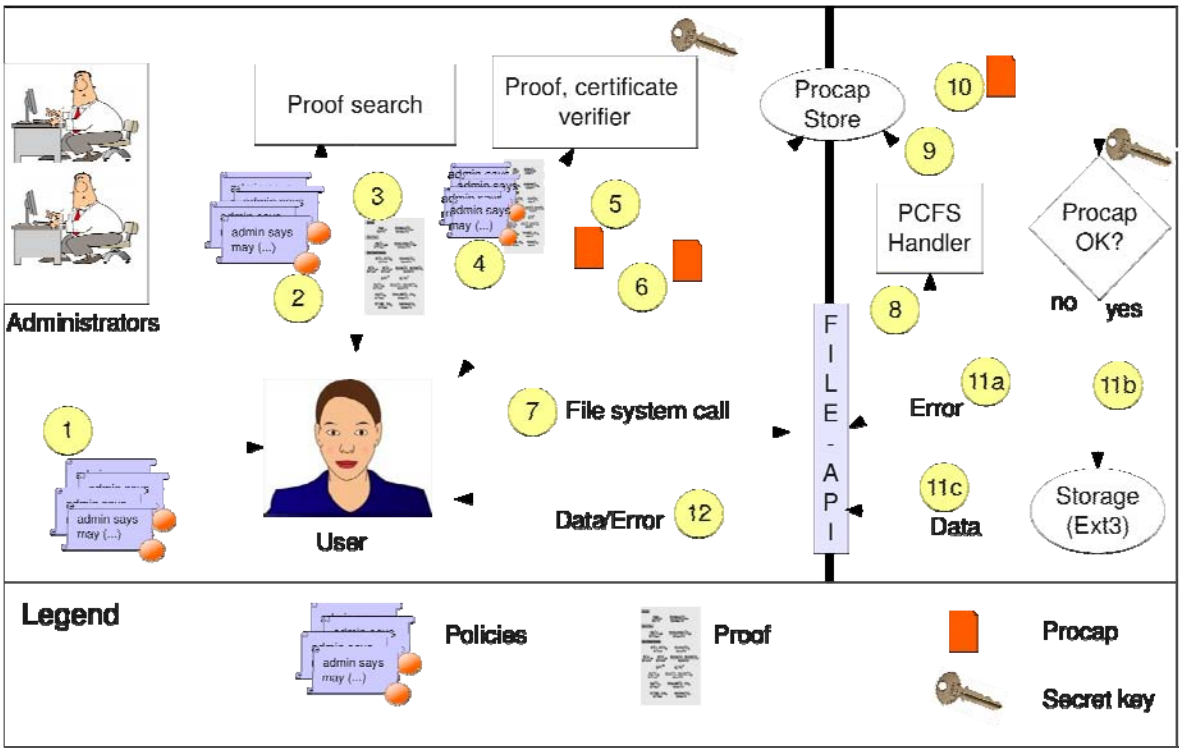

Figure 1: PCFS Architecture

Policy creation (Step 1). A policy is defined as a set of formulas in the logic BL that determine access rights. An access right is a triple <*k*, *f*, *and p*>, which means that user *k* (Alice, Bob, etc) has permission *p* (read, write, etc) on file or directory *f*. A policy is concretely represented as digital certificates, signed by individuals who create it. PCFS provides a command line tool, pcfs-cert, to help administrators check formulas for adherence to logical syntax, to digitally sign them, and to convert them to a custom certificate format.

Proof generation (Steps 2–3). Once certificates have been created by administrators and given to users, the latter use them to show that they are allowed certain permissions in the file system. The basic tenet of PCFS, as in PCA, is that a user *k* is allowed permission *p* on resource *f* at time *u*, if and only if the user can provide a *formal logical proof M* which shows that the policy in effect entails a fixed formula auth (*k*, *f*, *p*, *u*). To help users construct the proof *M*, PCFS provides an automatic theorem prover for BL, through the command line tool pcfs-search. This tool is based in logic programming; its underlying theory is explained in a technical report (Garg 2009). Figure 1 shows the user giving the policy (certificates) to the proof search tool in step 2, and the proof search tool returning a proof in step 3. Typical proof construction in PCFS takes several hundred milliseconds. A salient point is that the proof search tool is *not* a trusted component of PCFS and a user may use any method to create proofs.

Proof verification (Steps 4–5). Once the user has constructed a proof *M*, this proof (together with the certificates used to construct it) is given to a proof verifier, which is invoked using another command line program pcfs-verify (Step 4 in Figure 1). The verifier is a trusted component of PCFS. It checks that the logical structure of the proof *M* is correct, and that all certificates used in the proof are genuine, that is, their digital signatures check correctly. If both these hold, then the verifier gives back to the user a procap, which is a capability that mentions the right <*k*, *f*, *p*> that the proof grants (Step 5). The procap also contains some conditions on which the proof depends and is signed using a shared symmetric key that is known only to the verifier and the file system interface. The method used for verification of BL proofs and extraction of conditions from them is discussed in a technical report (Garg & Pfenning 2009). A typical proof verification including creation of a procap takes several tens or a few hundred milliseconds, depending on the size of the proof.

Procap injection (Step 6). After receiving a procap, the user invokes another command line tool to put the procap in a central store marked “Procap Store” in Figure 1. This store is in a designated part of the PCFS file system, and is accessible to both users as well as the system interface. The system interface looks up this store to find relevant procaps when file system calls are made.

File system call (Step 7). A call to the PCFS file system is made through the usual POSIX file system API during the execution of a user program or through a shell command. PCFS respects the standard POSIX interface, so user programs and shell commands don't need to change to work on it. However, before a file system call is executed the user or the program must ensure that procaps authorizing all needed permissions have been created and injected using Steps 2–6.

Procap look-up and checking (Steps 8–10). Once a program has made a file system call the file system looks up one or more procaps to authorize the operation (Steps 9 and 10). The exact procaps needed to authorize each operation vary. If all relevant procaps are found, they are checked. Checking a typical procap takes only 10–100μs (compare the time taken to check a proof, which is of the order of tens or hundreds of milliseconds.

Error (Steps 11a, 12). If any procap needed for performing the requested file operation is missing or fails to check an error code is returned to the user program.

File operation (Steps 11b, 11c, 12). If all relevant procaps needed to perform the requested file operation are found, and successfully check, then the file operation is performed. In the current implementation of PCFS, actual I/O is performed by redirecting to an existing file system (Step 11b). Hence PCFS is a *virtual file system* that layers logic-based access control on another file system.

4.4 Assessment of the Proof-Carrying File System

The merits of the logic-based approach to authorization have already been discussed above, as has its suitability for access-control policies in effect in the intelligence community. The more traditional approach to exploiting policies expressed in authorization logic has been *proof-carrying authorization* (PCA), where the proof of access is supplied upon request by the reference monitor, directly at time of access. There are two main advantages of the PCFS approach over more traditional architectures for PCA.

Modularity. Owing to the separation of the proof verifier from the reference monitor, the access control subsystem factors into two parts, both conceptually and in the implementation: (a) the *front end* that understands the logic and digital certificates, and performs proof search and proof verification to generate procaps, and (b) the *back end* that checks procaps to authorize access and performs resource access. The two parts only interact through procaps and are otherwise independent. In Figure 1, the front end corresponds to steps 1–6 and the back end corresponds to steps 7–12. This factorization has the following merits:

- The front end may be changed to support a different logic, or even replicated to support two authorization logics simultaneously, without any need to change the back end.
- The same front end can be used with different back ends
- The front end and back end can be implemented, tested, and debugged separately, possibly by different teams having expertise in logic and systems programming, respectively. There are no compile-time dependencies between the two parts, but they have to agree on a common structure for procaps.

Backwards compatibility. By storing procaps in a central location ("procap store" in Figure 1) rather than requiring programs to provide them at the time of access, as PCA does for proofs, PCFS is able to maintain backwards compatibility with the POSIX file system interface. This allows existing programs to run without modification, provided that enough procaps are generated in advance to authorize all access they need. A complication arises for files that programs create while they execute, in particular, temporary files that word processors and spreadsheets often create. To allow programs to access such files without the need to create and check proofs, the file system automatically generates default procaps that give the creating user read and write access to a new file or directory for a certain period of time. As a result, even sophisticated software like word processors and spreadsheets work seamlessly on PCFS. Access through default procaps can be turned off by changing an extended attribute on the file or directory on which such access is conditional.

4.5 Performance Evaluation of the Proof-Carrying File System Back End

In this section, a performance evaluation of the PCFS back end is presented. Specifically, microbenchmarks are used to evaluate the overhead of access checks during read, write, stat, create, and delete operations, together with a measurement of the effectiveness of an in-memory procap cache. To evaluate performance in practice, we also present the results of two simple macrobenchmarks. Since we are primarily interested in measuring the overhead of procap-based access checks, our baseline for comparing performance is a Fuse-based file system that does not perform the corresponding checks, but otherwise runs a server process and uses an underlying ext3 file system, just as PCFS does. We call this file system Fuse/Null. For macrobenchmarks we also compare with a native ext3 file system. All measurements reported here were made on a 2.4GHz Core Duo 2 machine with 3GB RAM and a 7200RPM 100GB hard disk drive, running the Linux kernel 2.6.24-23.

Read and write throughput. By default, PCFS does not make any access checks when read or write operations are performed on a previously opened file. As a result, it's read and writes throughput is very close to that of Fuse/Null. The following table summarizes the read and writes throughputs of PCFS and Fuse/Null based on reading and writing a 1GB file sequentially using the Bonnie++ test suite.

Table 1. Read and write throughput of PCFS

Operation	PCFS (MB/s)	Fuse/Null (MB/s)
Read	538.69	567.47
Write	73.18	76.05

Even if access checks on every read and write are enabled, read and write throughputs do not show a significant change as long as required procaps remain cached in memory.

File stats and effectiveness of caching. Besides read and write, two other very common file operations and open and stat (reading a file's meta-data). In terms of access checks, both are similar, since usually one procap must be checked in each case. We report in the table below the speed of the stat operation and the effect of the in-memory procap cache with different hit rates. All measurements are reported in number of operations per second, as well as time taken per operation. For comparison, performance of Fuse/Null is also shown.

As can be seen from Table 2, the procap cache is extremely helpful in attaining efficiency.

Table 2. Effectiveness of procap caching in PCFS

Cache hit rate	0%	50%	90%	95%	98%	100%	Fuse/Null
Stats per second	5774	7186	8871	9851	11879	23652	36042
Time per stat (μs)	163.2	139.2	112.7	101.5	84.2	42.2	27.7

File creation and deletion. Table 3 lists the number of create and delete operations per second that are supported by PCFS and Fuse/Null. These are measured by creating and deleting 10,000 files in a single directory.

Table 3. File creation and deletion in PCFS

Operation	PCFS (op/s)	Fuse/Null (op/s)
Create	1386	4738
Delete	1989	15429

PCFS is approximately 3.5 times slower than FUSE/Null in creating files. This is because in this experiment PCFS also created six default procaps for every file created. As a result, the PCFS numbers measure creation of seven times as many files in three separate directories. Deletion in PCFS in this experiment is nearly 7.7 times slower than that in Fuse/Null. This is because when a file is deleted in PCFS, one procap must be looked up, parsed, and checked, and all procaps related to the file must later be deleted. In this case, each file deletion in PCFS corresponds to seven file deletions on the ext3 file system in three different directories.

Macrobenchmarks. To understand the performance of PCFS in practice, we also ran two simple macrobenchmarks. The first (called OpenSSL in the table below), unpacks the OpenSSL source code, compiles it and deletes it. The other (called Fuse in the table below) performs similar operations for the source of the Fuse kernel module five times in sequence. As can be seen, the performance penalty for PCFS as compared to Fuse/Null is approximately 10% for OpenSSL, and 2.5% for Fuse. The difference arises because the OpenSSL benchmark depends more on file creation and deletion as compared to the Fuse benchmark. Ext3 is the native file system in the Linux kernel we used.

Table 4. PCFS Macrobenchmarks

Benchmark	PCFS	Fuse/Null	Ext3
OpenSSL	126	114	94
Fuse x 5	79	77	70

In summary, assuming a low rate of cache misses, the performance of PCFS on common file operations like read, write, stat, and open is comparable to that of Fuse/Null. On the other hand, less common operations like create and delete are slower because procaps must be managed.

4.6 Proof Search and Policy Analysis

Since logical proof is paramount to access control in the PCFS architecture, techniques for constructing such proofs from policies in the front end of PCFS and related systems have been developed. One approach is tailored to settings where credentials needed to complete a proof might need to be obtained from, or reactively created by, distant components in a distributed system. In such contexts, the approach substantially improves upon previous proposals in both computation and communication costs, and better guides users to create the most appropriate credentials in those cases where needed credentials do not yet exist. At the same time, the strategy offers strictly superior proving ability, in the sense that it finds a proof in every case that previous approaches would (and more). It has been implemented in the Grey access-control testbed at Carnegie Mellon, and further evaluated using simulations of other deployments (Bauer et al. 2007).

Proof-generation strategies have been further enhanced by showing that applying association rule mining to a history of accesses can predict changes to access-control policies that are likely to be consistent with users' intentions. This enables these changes to be instituted in advance of misconfigurations interfering with legitimate accesses, for example, by prompting the creation of needed credentials in a logic-based authorization framework. Instituting these changes requires consent of the appropriate administrator, of course, and so a primary contribution of this work is to automatically determine from whom to seek consent and to minimize the costs of doing so. We showed using data from the Grey deployment that these methods can reduce the number of accesses that would have incurred costly time-of-access delays by 43%, and can correctly predict 58% of the intended policy. These gains were achieved without impacting the total amount of time users spent interacting with the system (Bauer et al. 2008).

The techniques sketched above are not yet directly applicable in PCFS, which is currently implemented as a centralized system. Proof-search techniques in the centralized setting, for direct use in our PCFS file system prototype have also been developed in implemented. The resulting proof search procedure for a practically sufficient fragment of our logic BL (Garg 2009) is based on insights from the domain of logic programming, and has been shown to be sufficient for typical uses of PCFS (Chaudhuri & Garg 2009).

Finally, some techniques for reasoning about policies themselves, expressed in our authorization logic, have been developed. This requires reasoning about the semantic consequences of policies, specifically with respect to information flow as represented by the knowledge state of the principals. The preliminary results are quite promising (DeYoung & Pfenning 2009).

5 TECHNOLOGY TRANSITION

Symantec believes that aspects of the technology explored under this effort have transition potential. For this reason, Symantec has applied more than a dozen software developers to the task of building a commercially viable prototype over the course of a year.

6 CONCLUSIONS

Traditional methods for access control, such as access control lists, operate at a very low level of abstraction: with individual files and individual principals. This creates numerous problems in policy administration, because high-level policies about who should have access to which data have to be *manually* translated into access control lists and be maintained. This is both labor-intensive and error-prone, supported by much anecdotal evidence both from industry and the intelligence community. Ideally, access control policies can be both specified and enforced at a very high level of abstraction, namely at the level they are stated in informal government documents detailing the handling of sensitive information.

The key components to achieving this are logic *of authorization* that is expressive enough to capture the kinds of policies in effect in the intelligence community and a practical enforcement mechanism that directly uses the logic. The central accomplishments of this project can therefore be summarized as follows:

- Development of a logic rich enough to accurately capture policy motifs in the intelligence community. Briefly, access to a sensitive document is granted if there is a *formal proof* in the logic attesting this fact. This proof embodies the reason why a principal should be given this access, leading to greater accountability than hitherto possible.
- Case study of policies in effect in the intelligence community. The policies have been elicited through personal interviews of intelligence officials as well as publicly available reference material. The result is a formalization of a policy, which should be seen as an approximate model; consistent with unclassified information we were able to gather.
- Implementation of a file system that enforces access control policies through formal proof. This file system called PCFS has been evaluated with respect to efficiency goal and found to be practical for typical uses. Moreover, its architecture makes it backward compatible with POSIX standards, which eases the difficult of using it with standard software packages.

We conclude that security architecture for sensitive information, in particular in the intelligence community, is practical and can have numerous benefits, including reduction of error rates and greater accountability for information flow.

7 RECOMMENDATIONS

Based on our findings, we make two primary recommendations.

- Complex authorization policies regarding sensitive information in the intelligence community can be expressed in sufficiently rich authorization logic such as BL. The rendering of policies in logical form can be used as a fulcrum between policy administration on one side, and policy enforcement on the other, which is significantly less vulnerable to human error. Moreover, formal proofs underlying access control decisions have the potential to constitute raw material for powerful auditing tools. We recommend that questions of workflow, audit, and policy analysis be pursued further in order to address the remaining problems that have to do with the human interface to the developed technology.
- The prototype implementation of a proof-carrying file system (PCFS) has demonstrated that potential efficiency barriers can be overcome. PCFS is practical. However, in the intelligence community we believe there is a greater need for distributed secure storage as compared to a local file system like PCFS. This raises new opportunities and new questions, specifically with respect to certificate discovery, policy administration, and various protocol issues. A further investment in this technology seems indicated, in particular after the initial success of the technology transition effort at Symantec that explores some of these ideas.

8 REFERENCES

(Bauer et al. 2007)
Lujo Bauer, Scott Garriss, and Michael K. Reiter, "Efficient proving for practical distributed access-control systems," *12th European Symposium on Research in Computer Security*, Springer LNCS **4734,** September 2007 pp. 19–37.

(Bauer et al. 2008)
Lujo Bauer, Scott Garriss, and Michael K. Reiter, "Detecting and resolving policy misconfigurations in access-control systems," *13th ACM Symposium on Access Control Models and Technologies*, June 2008 pp. 19–37.

(Chaudhuri & Garg 2009)
Avik Chaudhuri and Deepak Garg, "Pcal: Language support for proof-carrying authorization systems," *European Symposium on Research in Computer Security (ESORICS'09)*, Saint Malo, France, September 2009.

(DeYoung et al. 2008)
Henry DeYoung, Deepak Garg, and Frank Pfenning. "An authorization logic with explicit time," *Proceedings of the 21st Computer Security Foundations Symposium (CSF-21)*, June 2008 pp 133–145
Extended version available as Technical Report CMU-CS-07-166, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, Pennsylvania, 15213, revised February 2008.

(DeYoung & Pfenning 2009)
Henry DeYoung and Frank Pfenning, "Reasoning about the consequences of authorization policies in a linear epistemic logic," *Workshop on Foundations of Computer Security (FCS'09)*, Los Angeles, California, August 2009.

(Garg 2008)
Garg, Deepak, "Principal-centric reasoning in constructive authorization logic," *Workshop on Intuitionistic Modal Logic and Applications (IMLA'08)*, July 2008.
Extended and revised version available as Technical Report CMU-CS-09-120, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, Pennsylvania, 15213, April 2009.

(Garg 2009)
Deepak Garg. *Proof search in an authorization logic*, CMU-CS-09-139, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, Pennsylvania, 15213, April 2009.

(Garg & Pfenning 2009)
Deepak Garg and Frank Pfenning, *A proof-carrying file system*, CMU-CS-09-123, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, Pennsylvania, 15213, June 2009.

(Garg et al. 2009)
Deepak Garg, Frank Pfenning, Denis Serenyi, and Brian Witten, *A logical representation of common rule for controlling access to classified information*, CMU-CS-09-139, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, Pennsylvania, 15213, June 2009.

(PCFS 2009)
PCFS, a proof-carrying file system.
Available at http://www.cs.cmu.edu/~dg/pcfs, Version 2.1, June 2009.

(Serenyi & Witten 2008a)
Denis Serenyi and Brian Witten, *Reports on policy formulation dynamics: Information access control policies with the intelligence community*, Internal Reports 1–5, Symantec Corporation, 20330 Stevens Creek Blvd., Cupertino, CA 95014, January–November 2008.

(Serenyi & Witten 2008b)
Denis Serenyi and Brian Witten, *Initial design of PCA file system and workflow control*, Internal report, Symantec Corporation, 20330 Stevens Creek Blvd., Cupertino, CA 95014, November 2008.

9 ACRONYMS/GLOSSARY

PCFS	Proof-Carrying File System
ACL	Access Control List
PCA	Proof-Carrying Authorization
BL	An authorization logic with explicit time and interpreted predicates
Procap	Proven Capability, signed by a trusted proof verify
Fuse	File System in User Space, basis for microbenchmark performance evaluation